DAM
Desarrollo de Aplicaciones Multiplataforma
2º Curso

AD
Acceso a Datos

UD 2
Manejo de Ficheros
Parte 2

IES BALMIS
Dpto Informática
Curso 2019-2020
Versión 1 (09/2019)

UD2 – Manejo de Ficheros

ÍNDICE

5. Ficheros de Texto - Escritura

Java incluye muchas clases relacionadas con la entrada y salida en fichero. No veremos todas ellas, sino solo las que según nuestro criterio pueden resultar más útiles y más sencillas.

Un fichero de texto se puede manejar de forma muy similar a la consola, volcando datos línea a línea con "**println**". Una de las formas más simples de conseguirlo en Java es empleando un "**PrintWriter**".

A diferencia de otros lenguajes como C#, es obligatorio usar un try-catch para interceptar los posibles errores (al menos FileNotFoundException, que se puede reemplazar por IOException, más general):

```
import java.io.PrintWriter;
import java.io.FileNotFoundException;

public class PrintWriter1 {
  public static void main(String[] args) {
    try {
      PrintWriter printWriter = new PrintWriter("ejemplo.txt");
      printWriter.println("Hola!");
      printWriter.close ();
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    }
  }
}
```

Una alternativa menos elegante, pero más compacta, por lo que puede ser útil en fuentes de pequeño tamaño, es no interceptar la excepción, sino permitir que el propio “main” pueda lanzarla, añadiendo “throws”, así:

```
import java.io.PrintWriter;
import java.io.FileNotFoundException;

public class PrintWriter1b {
  public static void main(String[] args) throws FileNotFoundException {
    PrintWriter printWriter = new PrintWriter ("ejemplo1.txt");
    printWriter.println ("Hola!");
    printWriter.close ();
  }
}
```

Siendo estrictos, puede existir un problema con ese primer programa: si se produce un error durante la escritura, el fichero podría quedar abierto y existir una pequeña fuga de memoria. Para evitarlo, es preferible cerrar el fichero en el bloque final opcional de "**try-catch-finally**".

De paso, podemos interceptar cualquier error de entrada / salida con **IOException**, e importar todo lo relacionado con la entrada y salida haciendo "**import java.io.*;**" en vez de emplear varios "**import**" individuales:

```
import java.io.*;

public class PrintWriter2 {
  public static void main(String[] args) {

    PrintWriter printWriter = null;
    try {
      printWriter = new PrintWriter("ejemplo2.txt");
      printWriter.println("Hola!");
      printWriter.println("y...");
      printWriter.println("hasta luego!");
    } catch (IOException e) {
      e.printStackTrace();
    } finally {
      if ( printWriter != null ) {
        printWriter.close();
      }
    }

  }
}
```

El constructor habitual de PrintWriter destruye el contenido fichero, en caso de que éste existiera. Si, por el contrario, se desea añadir al final de un fichero existente, se debe emplear otro constructor, que no recibirá la ruta del fichero, sino un "**BufferedWriter**", que se apoya en un "**FileWriter**" que tiene "**true**" como segundo parámetro: "**new FileWriter("ejemplo.txt", true))**", así:

```
import java.io.*;

public class PrintWriter3 {
  public static void main(String[] args) {
    PrintWriter printWriter = null;
    try {
      printWriter = new PrintWriter(new BufferedWriter(
          new FileWriter("ejemplo3.txt", true)));
      printWriter.println("Hola otra vez!");
      printWriter.println("y...");
      printWriter.println("hasta luego!");
    } catch (IOException e) {
      e.printStackTrace();
    } finally {
      if ( printWriter != null ) {
        printWriter.close();
      }
    }
  }
}
```

También se podría haber empleado directamente un "**BufferedWriter**" para escribir, sin necesidad de usar la clase "**PrintWriter**", pero se trata de una clase de más bajo nivel, que se parece menos al manejo habitual de la consola, porque no incluye un método "**println**", sino que se debe usar "**write**" para escribir y "**newLine**" para avanzar de línea, como muestra este ejemplo básico:

```
import java.io.*;

public class BufferedWriter1 {
  public static void main(String[] args) throws IOException {
    BufferedWriter ficheroSalida = null;
    try {
      ficheroSalida = new BufferedWriter(
          new FileWriter(new File("ejemplo4.txt")));
      ficheroSalida.write("Línea 1");
      ficheroSalida.newLine();
      ficheroSalida.write("Línea 2");
      ficheroSalida.newLine();
    } catch (IOException e) {
      System.out.println("Ha habido problemas: " + e.getMessage() );
    } finally {
      if ( ficheroSalida != null ) {
        ficheroSalida.close();
      }
    }
  }
}
```

6. Ficheros de Texto - Lectura

Una forma sencilla de leer de un fichero, si nos basta con hacerlo línea a línea, es emplear un "**BufferedReader**", que se apoyará en un "**FileReader**" y que incluye un método "**readLine**" que nos devuelve una cadena de texto (un "**String**").

Si ese String es null, quiere decir que se ha acabado el fichero y no se ha podido leer nada. Por eso, lo habitual es usar un "**while**" (o un "do-while") para leer todo el contenido de un fichero:

```
import java.io.*;

public class BufferedReader1 {
  public static void main(String[] args) {

    // Primero vemos si el fichero existe
    if (! (new File("ejemplo.txt")).exists() ) {
      System.out.println("No he encontrado ejemplo.txt");
      return;
    }

    // En caso de que exista, intentamos leer
    System.out.println("Leyendo fichero...");

    try {
      BufferedReader ficheroEntrada = new BufferedReader(
           new FileReader(new File("ejemplo.txt")));

      String linea = null;

      linea = ficheroEntrada.readLine();
      while (linea != null) {
        System.out.println(linea);
        linea = ficheroEntrada.readLine();
      }
      ficheroEntrada.close();

    } catch (IOException e) {
      System.out.println( "Ha habido problemas: " + e.getMessage() );
    }

    System.out.println("Fin de la lectura.");
  }
}
```

Es habitual abreviar la lectura y la comprobación, haciéndolas en una misma orden, lo que resulta más compacto pero menos legible:

```
...
  String linea=null;
  while ((linea=ficheroEntrada.readLine()) != null) {
    System.out.println(linea);
  }
...
```

7. Ficheros Binarios - Lectura

Un fichero "binario" es un fichero que contiene "cualquier cosa", no solo texto.

Se puede acceder a ellos con un fichero de tipo "**FileInputStream**" y leer byte a byte con "**read()**". El dato se deberá leer como "int", y un valor de "-1" indicará que se ha alcanzado el final del fichero.

Este ejemplo cuenta la cantidad de letras "a" que contiene un fichero de cualquier tipo, ya sea de texto, ejecutable, documento creado con un procesador de textos o cualquier otro fichero.

```
import java.io.*;

public class FileInputStream1 {
  public static void main(String[] args) {

    System.out.println("Contando \"a\"...");
    int contador = 0;

    try {

      FileInputStream ficheroEntrada2 = new FileInputStream(
          new File("fichero.bin"));
      int dato = ficheroEntrada2.read();
      while (dato != -1) {
        if (dato == 97) { // Codigo ASCII de "a"
          contador++;
        }
        dato = ficheroEntrada2.read();
      }
      ficheroEntrada2.close();

    } catch (Exception e) {
      System.out.println( "Ha habido problemas: " + e.getMessage() );
    }

    System.out.println("Cantidad de \"a\": " + contador);

  }
}
```

Nuevamente, también se puede abreviar la lectura y la comprobación, así:

```
...
  int dato;
  while ((dato = ficheroEntrada2.read()) != -1) {
    if (dato == 97) // Codigo ASCII de "a"
      contador++;
  }
...
```

8. Ficheros Binarios – Lectura y escritura por bloques

Si es necesario leer muchos datos de un fichero de cualquier tipo, acceder byte a byte puede resultar muy lento.

Una alternativa mucho más eficiente es usar un **array de bytes**. En este caso, se empleará un "**InputStream**", y su método "**read**", indicándole en qué array se desea guardar los datos, desde qué posición del array (que casi siempre será la cero) y qué cantidad de datos.

Si no conseguimos leer tantos datos como hemos intentado, será porque hemos llegado al final del fichero. Por eso, un programa que duplicara ficheros, leyendo cada vez un bloque de 512 Kb podría ser:

```
import java.io.*;

public class FicheroBinarioBloques {
  public static void main(String[] args) {

    System.out.println("Copiando fichero binario...");
    final int BUFFER_SIZE = 512*1024;
    try {
      InputStream ficheroEntrada3 = new FileInputStream( new File("fichero.in"));
      OutputStream ficheroSalida3 = new FileOutputStream(new File("fichero2.out"));
      byte[] buf = new byte[BUFFER_SIZE];
      int cantidadLeida;
      while ((cantidadLeida = ficheroEntrada3.read(buf, 0, BUFFER_SIZE)) > 0) {
        ficheroSalida3.write(buf, 0, cantidadLeida);
      }
      ficheroEntrada3.close();
      ficheroSalida3.close();
    } catch (Exception errorDeFichero) {
      System.out.println( "Ha habido problemas: " + errorDeFichero.getMessage() );
    }
    System.out.println("Terminado!");

  }
}
```

9. Serialización

Guardar un objeto en un fichero no es mucho más difícil que guardar datos nativos.

Por una parte, la clase correspondiente deberá implementar la interfaz "**Serializable**".

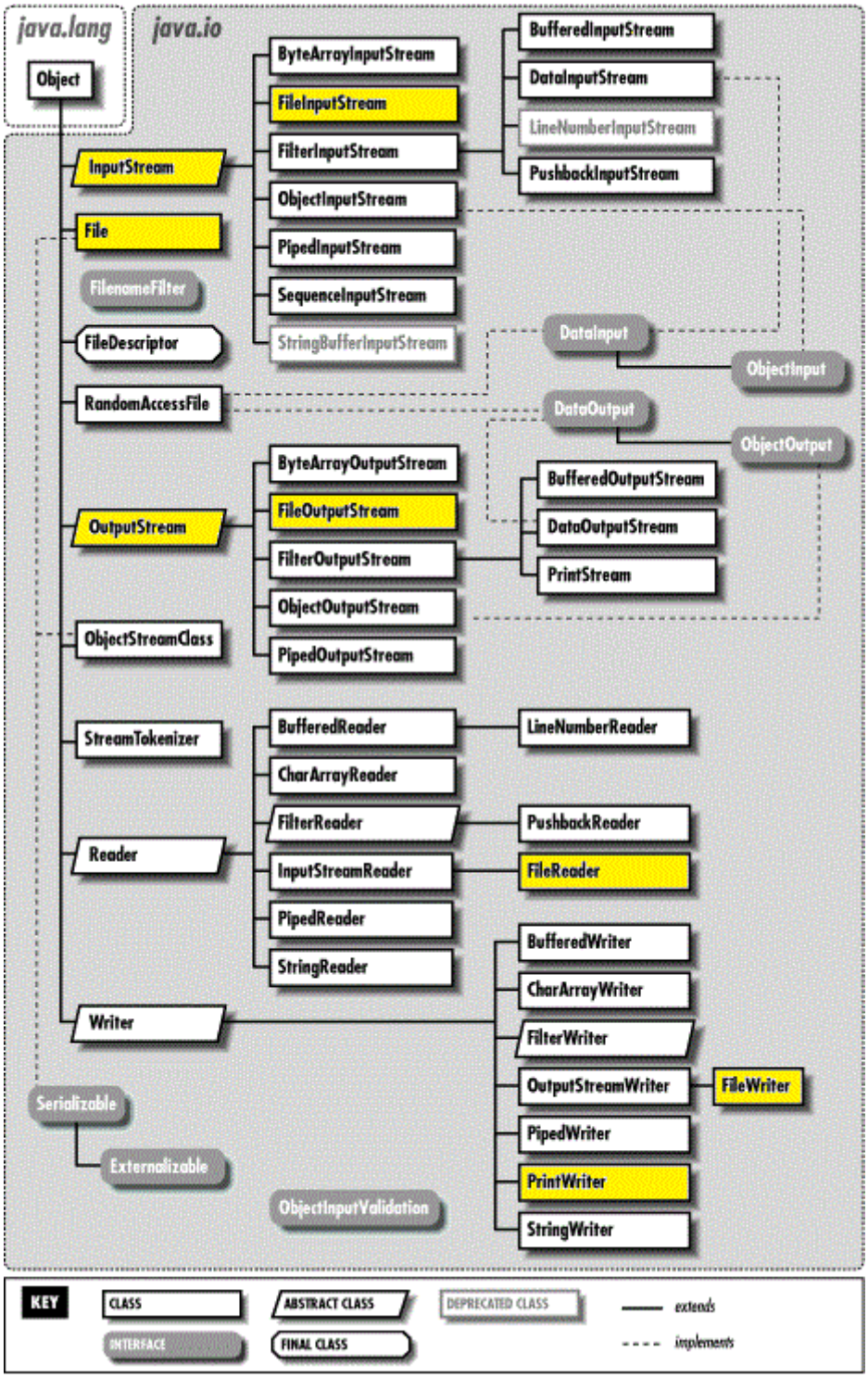
Por otra parte, los ficheros deberán ser de tipo **ObjectOutputStream** y **ObjectInputStream.**

El modo de leer es similar, pero empleando el método **readObject** de la clase ObjectInputStream e interceptando (o lanzando) la excepción ClassNotFoundException:

```
import java.io.*;

class Ciudad implements Serializable {
  protected String codigo;
  protected String nombre;

  // Constructor
  public Ciudad(String cod, String nom) {
    codigo = cod;
    nombre = nom;
  }

  // Método adicional
  public void escribir() {
    System.out.println(codigo + ": " + nombre);
  }
}

public class Serializar {
  public static void main(String[] args)
        throws FileNotFoundException, IOException, ClassNotFoundException {

    // Escribir - Serializar
    File fichero1 = new File("ciudades.dat");
    FileOutputStream ficheroSalida = new FileOutputStream(fichero1);
    ObjectOutputStream ficheroObjetos1 = new ObjectOutputStream(ficheroSalida);

    Ciudad c = new Ciudad("A", "Alicante");
    ficheroObjetos1.writeObject(c);

    c = new Ciudad("Gr", "Granada");
    ficheroObjetos1.writeObject(c);

    ficheroObjetos1.close();

    // Leer - Desserializar
    File fichero2 = new File("ciudades.dat");
    FileInputStream ficheroEntrada = new FileInputStream(fichero2);
    ObjectInputStream ficheroObjetos2 = new ObjectInputStream(ficheroEntrada);

    while (ficheroEntrada.available() > 0) {
      c = (Ciudad) ficheroObjetos2.readObject();
      c.escribir();
    }

    ficheroObjetos.close();
  }
}
```

10. Java.io

Hemos usado varios objetos de Java para trabajar con ficheros, pero existen muchos más.



Para ver más ejemplos
https://www.javatpoint.com/java-io